C++ Programming

Learning verse

-Happy learning

Introduction

C++ is a general-purpose, free-form programming language that supports object-oriented and generic programming. It was developed by Bjarne Stroustrup in 1979 at Bell Labs. It runs on a variety of platforms such as Windows, Mac OS and various versions of UNIX. It is the most widely used programming language in application and system programming.

Applications of C++ Programming

- Banking Applications
- Embedded systems
- GUI based Applications
- Web browsers
- Compilers

Difference between C and C++

C	C++
C is structure/procedure oriented programming language	C++ is object oriented programming language
C program design is top down approach	C++ is using bottom up approach
Polymorphism, virtual function, inheritance , Operator overloading , namespace concepts are not available in C programming language	C++ supports all these concepts and features
C language gives importance to functions rather than data	C++ gives importance to data rather than functions
So, data and function mapping is difficult in C	But, data and function mapping is simple in C++ that can be done using objects
C does not support user define data types	C++ supports user define data types
Exception handling is not present in C	Exception handling present in C++

Syntax

```
#include <iostream>          //Line 1
using namespace std;         //Line 2
//Line 3
int main() {                 //Line 4
  cout << "Hello World!";    //Line 5
  return 0;                  //Line 6
}                            //Line 7
```

Line 1: #include <iostream> is a **header file library** that lets us work with input and output objects, such as cout (used in line 5). Header files add functionality to C++ programs.
Line 2: using namespace std means that we can use names for objects and variables from the standard library.
Line 3: A blank line. C++ ignores white space. But we use it to make the code more readable.
Line 4: Another thing that always appear in a C++ program, is int main(). This is called a **function**. Any code inside its curly brackets {} will be executed.
Line 5: cout (pronounced "see-out") is an **object** used together with the *insertion operator* (<<) to output/print text. In our example it will output "Hello World".

Note: Every C++ statement ends with a semicolon ;.
Note: The body of int main() could also been written as:
int main () { cout << "Hello World! "; return 0; }
Remember: The compiler ignores white spaces. However, multiple lines makes the code more readable.

Line 6: return 0 ends the main function.
Line 7: Do not forget to add the closing curly bracket } to actually end the main function.

C++ Comments

Comments can be used to explain C++ code, and to make it more readable. It can also be used to prevent execution when testing alternative code. Comments can be singled-lined or multi-lined.

Single-line Comments

Single-line comments start with two forward slashes (//).

Any text between // and the end of the line is ignored by the compiler (will not be executed).

This example uses a single-line comment before a line of code:

```
#include <iostream>
using namespace std;

int main() {
  // This is a comment
  cout << "Hello World!";
  return 0;
}
```

OUTPUT

```
Hello World!
```

C++ Multi-line Comments

Multi-line comments start with /* and ends with */.

Any text between /* and */ will be ignored by the compiler

Ex:

```
#include <iostream>
using namespace std;

int main() {
  /* The code below will print the words Hello World!
  to the screen, and it is amazing */
  cout << "Hello World!";
  return 0;
}
```

OUTPUT

```
Hello World!
```

C++ Variables

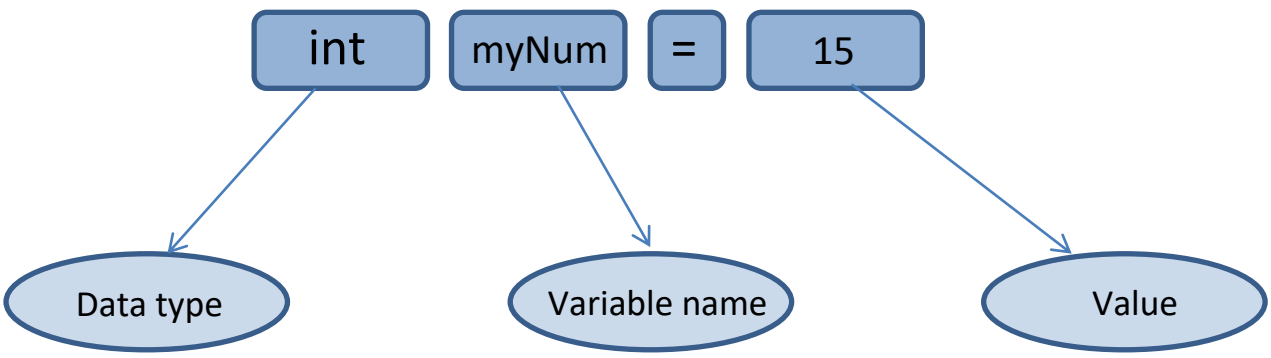
Variables are containers for storing data values.
In C++, there are different types of variables (defined with different keywords), for example:

- **int** - stores integers (whole numbers), without decimals, such as 123 or -123
- **double** - stores floating point numbers, with decimals, such as 19.99 or -19.99
- **char** - stores single characters, such as 'a' or 'B'. Char values are surrounded by single quotes
- **string** - stores text, such as "Hello World". String values are surrounded by double quotes
- **bool** - stores values with two states: true or false

Declaring (Creating) Variables

To create a variable, specify the type and assign it a value:
Syntax:
type variableName = value*;*
Ex:

```
#include <iostream>
using namespace std;
int main() {
  int myNum = 15;
  cout << myNum;
  return 0;
}
```



C++ Identifiers

All C++ **variables** must be **identified** with **unique names**.
These unique names are called **identifiers**.
Identifiers can be short names (like x and y) or more descriptive names (age, sum, totalVolume).
Note: It is recommended to use descriptive names in order to create understandable and maintainable code:
Ex:

```
#include <iostream>
using namespace std;
int main() {
  // Good name
  int minutesPerHour = 60;

  // OK, but not so easy to understand what m actually is
  int m = 60;

  cout << minutesPerHour << "\n";
  cout << m;
  return 0;
}
```

OUTPUT

```
60
60
```

The general rules for naming variables are:

- Names can contain letters, digits and underscores
- Names must begin with a letter or an underscore (_)
- Names are case sensitive (myVar and myvar are different variables)
- Names cannot contain whitespaces or special characters like !, #, %, etc.
- Reserved words (like C++ keywords, such as int) cannot be used as names

Constants

When you do not want others (or yourself) to override existing variable values, use the const keyword (this will declare the variable as "constant", which means **unchangeable and read-only**):

```
#include <iostream>
using namespace std;
int main() {
  const int myNum = 15;
  myNum = 10;
  cout << myNum;
  return 0;
}
```

OUTPUT

```
error: assignment of read-only variable ‘myNum’
```

You should always declare the variable as constant when you have values that are unlikely to change:

Ex:

```
#include <iostream>
using namespace std;
int main() {
  const int minutesPerHour = 60;
  const float PI = 3.14;
  cout << minutesPerHour << "\n";
  cout << PI;
  return 0;
}
```

OUTPUT

```
60
3.14
```

C++ Data Types

```
#include <iostream>
#include <string>
using namespace std;

int main () {
  // Creating variables
  int myNum = 5;               // Integer (whole number)
  float myFloatNum = 5.99;     // Floating point number
  double myDoubleNum = 9.98;   // Floating point number
  char myLetter = 'D';         // Character
  bool myBoolean = true;       // Boolean
  string myString = "Hello";   // String

  // Print variable values
  cout << "int: " << myNum << "\n";
  cout << "float: " << myFloatNum << "\n";
  cout << "double: " << myDoubleNum << "\n";
  cout << "char: " << myLetter << "\n";
  cout << "bool: " << myBoolean << "\n";
  cout << "string: " << myString << "\n";

  return 0;
}
```

OUTPUT

```
int: 5
float: 5.99
double: 9.98
char: D
bool: 1
string: Hello
```

Basic Data Types

The data type specifies the size and type of information the variable will store:

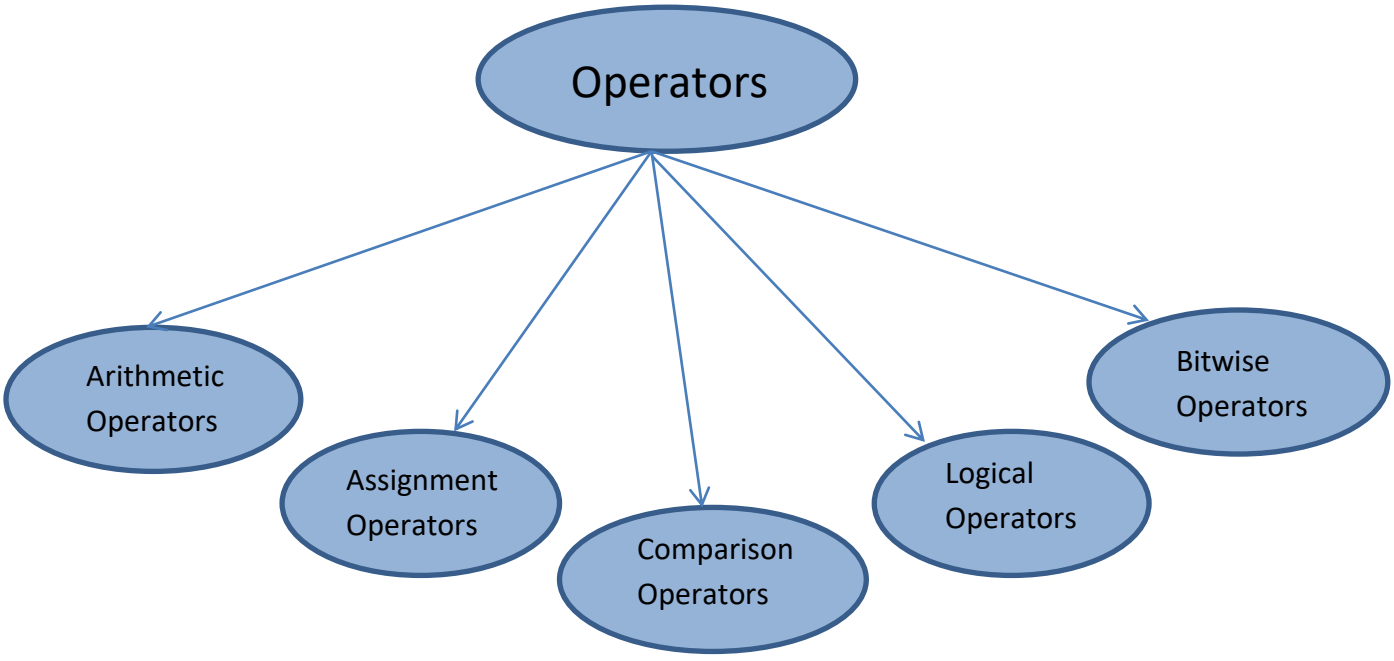
Data Type	Size	Description
boolean	1 byte	Stores true or false values
char	1 byte	Stores a single character/letter/number, or ASCII values
int	2 or 4 bytes	Stores whole numbers, without decimals
float	4 bytes	Stores fractional numbers, containing one or more decimals. Sufficient for storing 7 decimal digits
double	8 bytes	Stores fractional numbers, containing one or more decimals. Sufficient for storing 15 decimal digits

C++ Operators

Operators are used to perform operations on variables and values.
In the example below, we use the + **operator** to add together two values:
It can also be used to add together a variable and a value, or a variable and another variable:
Ex:

```
#include <iostream>
using namespace std;
int main() {
  int sum1 = 100 + 50;       // 150 (100 + 50)
  int sum2 = sum1 + 250;     // 400 (150 + 250)
  int sum3 = sum2 + sum2;    // 800 (400 + 400)
  cout << sum1 << "\n";
  cout << sum2 << "\n";
  cout << sum3;
  return 0;
}
```

OUTPUT
150
400
800



Arithmetic Operators

Arithmetic operators are used to perform common mathematical operations.

Operator	Name	Description	Example
+	Addition	Adds together two values	x + y
-	Subtraction	Subtracts one value from another	x - y
*	Multiplication	Multiplies two values	x * y
/	Division	Divides one value by another	x / y
%	Modulus	Returns the division remainder	x % y
++	Increment	Increases the value of a variable by 1	++x
--	Decrement	Decreases the value of a variable by 1	--x

Assignment Operators

Assignment operators are used to assign values to variables.
In the example below, we use the **assignment** operator (=) to assign the value **10** to a variable called **x**:

```
#include <iostream>
using namespace std;
int main() {
  int x = 10;
  x += 5;
  cout << x;
  return 0;
}
```

OUTPUT

15

Comparison Operators

Comparison operators are used to compare two values (or variables). This is important in programming, because it helps us to find answers and make decisions.
The return value of a comparison is either 1 or 0, which means **true** (1) or **false** (0). These values are known as **Boolean values**, and you will learn more about them in the Booleans and If..Else topic
In the following example, we use the **greater than** operator (>) to find out if 5 is greater than 3:
Ex:

```
#include <iostream>
using namespace std;
int main() {
  int x = 5;
  int y = 3;
  cout << (x > y); // returns 1 (true) because 5 is greater than 3
  return 0;
}
```

C++ Conditions

The if Statement

Use the if statement to specify a block of C++ code to be executed if a condition is true.

Syntax:

```
if (condition) {
  // block of code to be executed if the condition is true
}
```

Ex:

```
#include <iostream>
using namespace std;
int main() {
  if (20 > 18) {
    cout << "20 is greater than 18";
  }
  return 0;
}
```

OUTPUT

```
20 is greater than 18
```

The else Statement

Use the else statement to specify a block of code to be executed if the condition is false.

Syntax:

```
if (condition) {
  // block of code to be executed if the condition is true
} else {
  // block of code to be executed if the condition is false
}
```

Ex:

```
#include <iostream>
using namespace std;
int main() {
  int time = 20;
  if (time < 18) {
```

```
  cout << "Good day.";
 } else {
  cout << "Good evening.";
 }
 return 0;
}
```

OUTPUT: Good evening

The else if Statement

Use the else if statement to specify a new condition if the first condition is false.

Syntax:

```
if (condition1) {
 // block of code to be executed if condition1 is true
} else if (condition2) {
 // block of code to be executed if the condition1 is false and condition2 is true
} else {
 // block of code to be executed if the condition1 is false and condition2 is false
}
```

Ex:

```
#include <iostream>
using namespace std;
int main() {
 int time = 22;
 if (time < 10) {
  cout << "Good morning.";
 } else if (time < 20) {
  cout << "Good day.";
 } else {
  cout << "Good evening.";
 }
 return 0;
}
```

OUTPUT: Good evening

C++ Loops

Loops can execute a block of code as long as a specified condition is reached.
Loops are handy because they save time, reduce errors, and they make code more readable.

While Loop

The while loop loops through a block of code as long as a specified condition is true:

Syntax:

```
while (condition) {
  // code block to be executed
}
```

Ex:

```
#include <iostream>
using namespace std;
int main() {
  int i = 0;
  while (i < 5) {
    cout << i << "  ";
    i++;
  }
  return 0;
}
```

OUTPUT

0 1 2 3 4

NOTE: Do not forget to increase the variable in the condition, otherwise the loop will never end!

The Do/While Loop

The do/while loop is a variant of the while loop. This loop will execute the code block once, before checking if the condition is true, then it will repeat the loop as long as the condition is true.

Syntax:

```
do {
  // code block to be executed
}
while (condition);
```

Ex:

```
int i = 0;
do {
  cout << i << "\n";
  i++;
}
while (i < 5);
```

C++ For Loop

When you know exactly how many times you want to loop through a block of code, use the for loop instead of a while loop:

Syntax:

```
for (statement 1; statement 2; statement 3) {
  // code block to be executed
}
```

Statement 1 is executed (one time) before the execution of the code block.

Statement 2 defines the condition for executing the code block.

Statement 3 is executed (every time) after the code block has been executed.

Ex:

```
#include <iostream>
using namespace std;
int main() {
  for (int i = 0; i < 5; i++) {
    cout << i << "  ";
  }
  return 0;
}
```

OUTPUT

0 1 2 3 4

C++ Arrays

Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value.
To declare an array, define the variable type, specify the name of the array followed by **square brackets** and specify the number of elements it should store:
To create an array of three integers:
int myNum[3] = {10, 20, 30};

Access the Elements of an Array

You access an array element by referring to the index number inside square brackets [].
This statement accesses the value of the **first element** in **myNum**:
Ex:
cout << myNum[0] << endl; //10
Note: Array indexes start with 0: [0] is the first element. [1] is the second element, etc.

Change an Array Element

To change the value of a specific element, refer to the index number:
myNum[1]=50

Array implementation

```
#include <iostream>
#include <string>
using namespace std;
int main() {
  string cars[4] = {"Volvo", "BMW", "Ford", "Mazda"};
  cars[0] = "Opel";
  cout << cars[0];
  return 0;
}
```

OUTPUT
Opel

Arrays using Loops

```
#include <iostream>
using namespace std;
int main() {
  int myNumbers[5] = {10, 20, 30, 40, 50};
  for (int i = 0; i < 5; i++) {
    cout << myNumbers[i] << "  ";
  }
  return 0;
}
```

OUTPUT

10 20 30 40 50

The foreach Loop

There is also a "**for-each** loop" (introduced in C++ version 11 (2011), which is used exclusively to loop through elements in an array:

Syntax:

```
for (type variableName : arrayName) {
  // code block to be executed
}
```

Ex:

```
#include <iostream>
using namespace std;
int main() {
  int myNumbers[5] = {10, 20, 30, 40, 50};
  for (int i : myNumbers) {
    cout << i << "\n";
  }
  return 0;
}
```

OUTPUT

10
20
30
40
50

C++ Structures

Structures (also called structs) are a way to group several related variables into one place. Each variable in the structure is known as a **member** of the structure.
Unlike an array, a structure can contain many different data types (int, string, bool, etc.).

Create a Structure

To create a structure, use the struct keyword and declare each of its members inside curly braces.
After the declaration, specify the name of the structure variable (**myStructure** in the example below):

Syntax:

```
struct {            // Structure declaration
  int myNum;        // Member (int variable)
  string myString;  // Member (string variable)
} myStructure;      // Structure variable
```

Access Structure Members

To access members of a structure, use the dot syntax (.):
Ex:

```
#include <iostream>
#include <string>
using namespace std;
int main() {
  struct {
    int myNum;
    string myString;
  } myStructure;
  myStructure.myNum = 1;
  myStructure.myString = "Hello World!";
  cout << myStructure.myNum << "\n";
  cout << myStructure.myString << "\n";
  return 0;
```

```
}
```

OUTPUT

```
1
Hello World!
```

One Structure in Multiple Variables

You can use a comma (,) to use one structure in many variables:

Syntax:

```
struct {
  int myNum;
  string myString;
} myStruct1, myStruct2, myStruct3; // Multiple structure variables separated with commas
```

Named Structures

By giving a name to the structure, you can treat it as a data type. This means that you can create variables with this structure anywhere in the program at any time.

To create a named structure, put the name of the structure right after the struct keyword:

Ex:

```
struct myDataType { // This structure is named "myDataType"
  int myNum;
  string myString;
};
```

To declare a variable that uses the structure, use the name of the structure as the data type of the variable:

```
myDataType myVar;
```

C++ Functions

A function is a block of code which only runs when it is called.
You can pass data, known as parameters, into a function.
Functions are used to perform certain actions, and they are important for reusing code: Define the code once, and use it many times.

Create a Function

C++ provides some pre-defined functions, such as main(), which is used to execute code. But you can also create your own functions to perform certain actions.
To create (often referred to as *declare*) a function, specify the name of the function, followed by parentheses **()**:

Syntax:

```
void myFunction() {
  // code to be executed
}
```

- myFunction() is the name of the function
- void means that the function does not have a return value. You will learn more about return values later in the next chapter
- inside the function (the body), add code that defines what the function should do

Call a Function

Declared functions are not executed immediately. They are "saved for later use", and will be executed later, when they are called.
To call a function, write the function's name followed by two parentheses () and a semicolon ;
In the following example, myFunction() is used to print a text (the action), when it is called:
Ex:

```
#include <iostream>
using namespace std;
void myFunction() {
  cout << "I just got executed!";
}
```

```
int main() {
  myFunction();
  return 0;
}
```

OUTPUT

I just got executed!

Parameters and Arguments

Information can be passed to functions as a parameter. Parameters act as variables inside the function.

Parameters are specified after the function name, inside the parentheses. You can add as many parameters as you want, just separate them with a comma

Syntax:

```
void functionName(parameter1, parameter2, parameter3) {
  // code to be executed
}
```

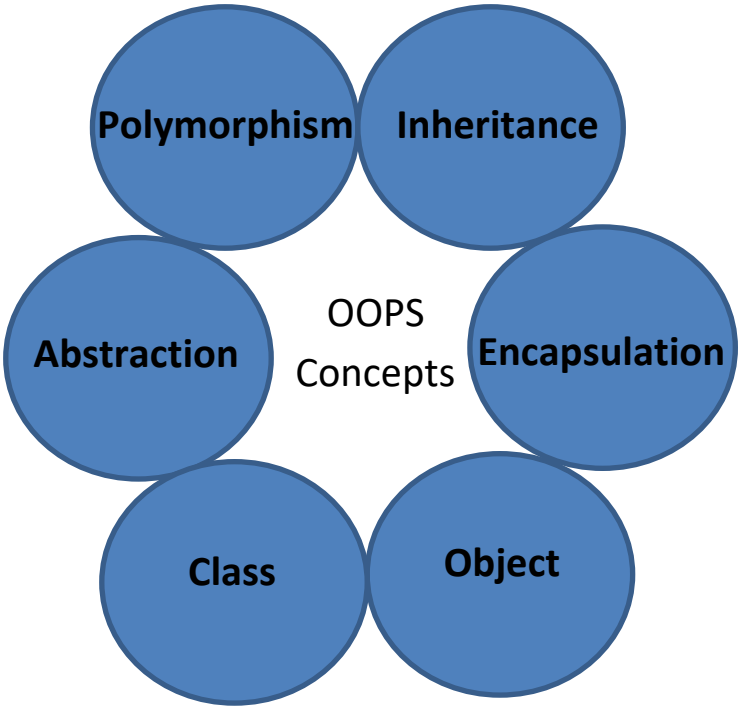
Ex:

```
#include <iostream>
#include <string>
using namespace std;
void myFunction(string fname) {
  cout << fname << " Refsnes\n";
}
int main() {
  myFunction("Liam");
  myFunction("Jenny");
  myFunction("Anja");
  return 0;
}
```

OUTPUT

Liam Refsnes
Jenny Refsnes
Anja Refsnes

Object Oriented Programming in C++



Object-oriented programming – As the name suggests uses objects in programming. Object-oriented programming aims to implement real-world entities like inheritance, hiding, polymorphism, etc in programming. The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function.

Class:

The building block of C++ that leads to Object-Oriented programming is a Class. It is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A class is like a blueprint for an object.

- A Class is a user-defined data-type which has data members and member functions.
- Data members are the data variables and member functions are the functions used to manipulate these variables and together these data members and member functions define the properties and behaviour of the objects in a Class.

- In the above example of class Car, the data member will be speed limit, mileage etc and member functions can apply brakes, increase speed etc.

Object:

An Object is an identifiable entity with some characteristics and behaviour. An Object is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.

Ex:

```
class person
{
        char name[20];
        int id;
public:
        void getdetails(){}
};
int main()
{
person p1; // p1 is a object
}
```

Object take up space in memory and have an associated address like a record in pascal or structure or union in C.

When a program is executed the objects interact by sending messages to one another.

Each object contains data and code to manipulate the data. Objects can interact without having to know details of each other's data or code, it is sufficient to know the type of message accepted and type of response returned by the objects.

Encapsulation

In normal terms, Encapsulation is defined as wrapping up of data and information under a single unit. In Object-Oriented Programming, Encapsulation is defined as binding together the data and the functions that manipulate them.

Consider a real-life example of encapsulation, in a company, there are different sections like the accounts section, finance section, sales section etc. The finance section handles all the financial transactions and keeps records of all the data related to finance. Similarly, the sales section handles all the sales-related activities and keeps records of all the sales. Now there may arise a situation when for some reason an official from the finance section needs all the data about sales in a particular month. In this case, he is not allowed to directly access the data of the sales section. He will first have to contact some other officer in the sales section and then request him to give the particular data. This is what encapsulation is. Here the data of the sales section and the employees that can manipulate them are wrapped under a single name “sales section”.

Encapsulation also leads to *data abstraction or hiding*. As using encapsulation also hides the data. In the above example, the data of any of the section like sales, finance or accounts are hidden from any other section.

Abstraction

Data abstraction is one of the most essential and important features of object-oriented programming in C++. Abstraction means displaying only essential information and hiding the details. Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation.

Consider a real-life example of a man driving a car. The man only knows that pressing the accelerators will increase the speed of the car or applying brakes will stop the car but he does not know about how on pressing accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of accelerator, brakes etc in the car. This is what abstraction is.

- **Abstraction using Classes**:
  We can implement Abstraction in C++ using classes. The class helps us to group data members and member functions using available access specifiers. A Class can decide which data member will be visible to the outside world and which is not.
- **Abstraction in Header files**:
  One more type of abstraction in C++ can be header files. For example, consider the pow() method present in math.h header file. Whenever we need to calculate the power of a number, we simply call the function pow() present in the math.h header file and pass the numbers as arguments without knowing the underlying algorithm according to which the function is actually calculating the power of numbers.

Polymorphism

The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.

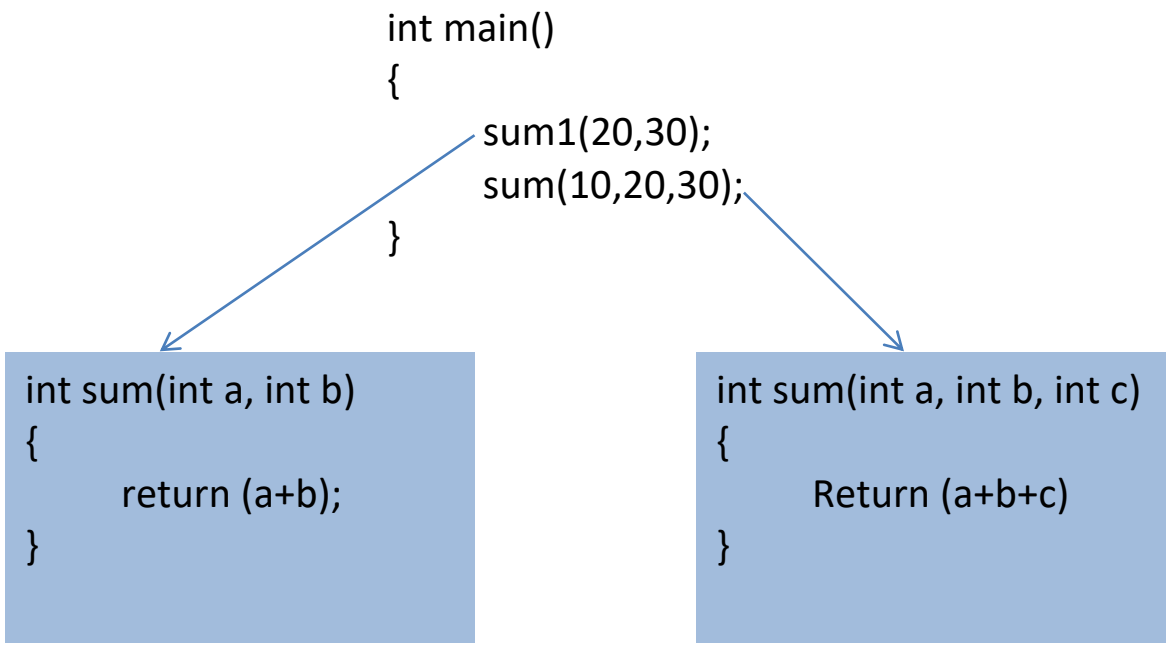
A person at the same time can have different characteristic. Like a man at the same time is a father, a husband, an employee. So the same person posses different behaviour in different situations. This is called polymorphism.

An operation may exhibit different behaviours in different instances. The behaviour depends upon the types of data used in the operation.

C++ supports operator overloading and function overloading.

- ***Operator Overloading***: The process of making an operator to exhibit different behaviours in different instances is known as operator overloading.
- ***Function Overloading***: Function overloading is using a single function name to perform different types of tasks.
  Polymorphism is extensively used in implementing inheritance.

Example: Suppose we have to write a function to add some integers, some times there are 2 integers, some times there are 3 integers. We can write the Addition Method with the same name having different parameters, the concerned method will be called according to parameters.



Inheritance

The capability of a class to derive properties and characteristics from another class is called Inheritance. Inheritance is one of the most important features of Object-Oriented Programming.

- **Sub Class**: The class that inherits properties from another class is called Sub class or Derived Class.
- **Super Class**:The class whose properties are inherited by sub class is called Base Class or Super class.
- **Reusability**: Inheritance supports the concept of “reusability”, i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

  Example: Dog, Cat, Cow can be Derived Class of Animal Base Class.